SPRING DEPENDENCY INJECTION

Every java based application has a few objects that work together to present what the end-user sees as a working application. When writing a complex Java application, application classes should be as independent as possible of other Java classes to increase the possibility to reuse these classes and to test them independently of other classes while doing unit testing. Dependency Injection (or sometime called wiring) helps in gluing these classes together and same time keeping them independent.

Consider you have an application which has a text editor component and you want to provide spell checking. Your standard code would look something like this:

```
public class TextEditor {
   private SpellChecker spellChecker;
   public TextEditor() {
      spellChecker = new SpellChecker();
   }
}
```

What we've done here is create a dependency between the TextEditor and the SpellChecker. In an inversion of control scenario we would instead do something like this:

```
public class TextEditor {
   private SpellChecker spellChecker;
   public TextEditor(SpellChecker spellChecker) {
      this.spellChecker = spellChecker;
   }
}
```

Here TextEditor should not worry about SpellChecker implementation. The SpellChecker will be implemented independently and will be provided to TextEditor at the time of TextEditor instantiation and this entire procedure is controlled by the Spring Framework.

Here, we have removed the total control from TextEditor and kept it somewhere else (ie. XML configuration file) and the dependency (ie. class SpellChecker) is being injected into the class TextEditor through a **Class Constructor**. Thus flow of control has been "inverted" by Dependency Injection (DI) because you have effectively delegated dependances to some external system.

Second method of injecting dependency is through **Setter Methods** of TextEditor class where we will create SpellChecker instance and this instance will be used to call setter methods to initialize TextEditor's properties.

Thus, DI exists in two major variants and following two sub-chapters will cover both of them with examples:

S.N.	Dependency Injection Type & Description
1	Constructor-based dependency injection Constructor-based DI is accomplished when the container invokes a class constructor with a number of arguments, each representing a dependency on other class.
2	Setter-based dependency injection Setter-based DI is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.

You can mix both, Constructor-based and Setter-based DI but it is a good rule of thumb to use constructor arguments for mandatory dependencies and setters for optional dependencies.

Code is cleaner with the DI principle and decoupling is more effective when objects are provided with their dependencies. The object does not look up its dependencies, and does not know the location or class of the dependencies rather everything is taken care by the Spring Framework.